CSE443
Compilers

Dr. Carl Alphonce
alphonce@buffalo.edu
343 Davis Hall
Phases of a compiler

Intermediate Representation (IR): specification and generation

Figure 1.6, page 5 of text
machine independent optimizations

IR

machine dependent optimizations

IR

target 1

target 2

... 

target n
Milestone 3

HLL

IR

Milestone 4

target 1

machine independent optimizations

Milestone 4 (maybe some simple ones)
Intermediate Representations
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]
Directed Acyclic Graph (DAG)

- Similar to a syntax tree
- No repeated nodes: structure sharing
Ex. 6.1 [p 359]

\[ a + a \cdot (b - c) + (b - c) \cdot d \]
Ex. 6.1 [p 359]

\[ a + a \times (b - c) + (b - c) \times d \]

Things can be more complicated if expressions have side effects.
SDT
Tree or DAG

<table>
<thead>
<tr>
<th>Production</th>
<th>Semantic Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 E → E₁ + T</td>
<td>E.node = new Node('+', E₁.node, T.node)</td>
</tr>
<tr>
<td>2 E → E₁ - T</td>
<td>E.node = new Node('-', E₁.node, T.node)</td>
</tr>
<tr>
<td>3 E → E₁ * T</td>
<td>E.node = new Node('*', E₁.node, T.node)</td>
</tr>
<tr>
<td>4 E → T</td>
<td>E.node = T.node</td>
</tr>
<tr>
<td>5 T → ( E )</td>
<td>T.node = E.node</td>
</tr>
<tr>
<td>6 T → id</td>
<td>T.node = new Leaf(id, id.entry)</td>
</tr>
<tr>
<td>7 T → num</td>
<td>T.node = new Leaf(num, num.val)</td>
</tr>
</tbody>
</table>

Figure 6.4 in text (p. 360), corrected according to errata sheet.
SDT

Tree or DAG

- SDT produces a tree if each call to Node creates a new tree node.

- SDT produces a DAG if for each call to Node there is a check whether this node already exists, and if so it returns a reference to the existing node rather than returning a new node.
### Example

<table>
<thead>
<tr>
<th>$p_1$</th>
<th>$\text{Leaf}(\text{id, entry-a})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_2$</td>
<td>$\text{Leaf}(\text{id, entry-a}) = p_1$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$\text{Leaf}(\text{id, entry-b})$</td>
</tr>
<tr>
<td>$p_4$</td>
<td>$\text{Leaf}(\text{id, entry-c})$</td>
</tr>
<tr>
<td>$p_5$</td>
<td>$\text{Node}('-', p_3, p_4)$</td>
</tr>
<tr>
<td>$p_6$</td>
<td>$\text{Node}('*', p_1, p_5)$</td>
</tr>
<tr>
<td>$p_7$</td>
<td>$\text{Node}('+', p_1, p_6)$</td>
</tr>
<tr>
<td>$p_8$</td>
<td>$\text{Leaf}(\text{id, entry-b}) = p_3$</td>
</tr>
<tr>
<td>$p_9$</td>
<td>$\text{Leaf}(\text{id, entry-c}) = p_4$</td>
</tr>
<tr>
<td>$p_{10}$</td>
<td>$\text{Node}('-', p_3, p_4) = p_5$</td>
</tr>
<tr>
<td>$p_{11}$</td>
<td>$\text{Leaf}(\text{id, entry-d})$</td>
</tr>
<tr>
<td>$p_{12}$</td>
<td>$\text{Node}('*', p_5, p_{11})$</td>
</tr>
<tr>
<td>$p_{13}$</td>
<td>$\text{Node}('+', p_7, p_{12})$</td>
</tr>
</tbody>
</table>
Value-number method
Algorithm 6.3 [p. 361]

Input: label op, node l, node r

Output: The value number of a node in the array with signature <op,l,r>

Method: Search the array for a node M with signature <op,l,r>. If there is such a node, return the value number of M. If not, create in the array a new node N with signature <op,l,r> and return its value number.
Value-number method
Algorithm 6.3 [p. 361]

- **Input:** label op, node l, node r
- **Output:** The value number of a node in the array with signature <op,l,r>
- **Method:** Search the array for a node M with signature <op,l,r>. If there is such a node, return the value number of M. If not, create in the array a new node N with signature <op,l,r> and return its value number.

Can use hash table for efficiency.
Revisiting 6.1
see construction steps in figure 6.5 [p. 360]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>id</td>
<td>→ to ST entry for a</td>
</tr>
<tr>
<td>2</td>
<td>id</td>
<td>→ to ST entry for b</td>
</tr>
<tr>
<td>3</td>
<td>id</td>
<td>→ to ST entry for c</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>+</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>id</td>
<td>→ to ST entry for d</td>
</tr>
<tr>
<td>8</td>
<td>*</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>+</td>
<td>6</td>
</tr>
</tbody>
</table>
Three-address code

- The DAG does not say anything about how the computation should be carried out.

- For example, there could be one instruction to do this computation:
  \[ t_1 = x + y * z \]
Three-address code

- In three-address code instructions can have no more than one operator on the right of an assignment.

- \(x + y \times z\) must be broken into two instructions:
  \[t_1 = y \times z\]
  \[t_2 = x + t_1\]
Three-address code representation

- $t_1 = b - c$
- $t_2 = a \times t_1$
- $t_3 = a + t_2$
- $t_4 = t_1 \times d$
- $t_5 = t_3 + t_4$

"Three-address code is a linearized representation of ... a DAG in which explicit names correspond to the interior nodes of the graph." [p. 363]
Three address code instructions
(see 6.2.1, pages 364-5)

1. \( x = y \text{ op } z \)
2. \( x = \text{ op } y \) (treat i2r and r2i as unary ops)
3. \( x = y \)
4. \( \text{goto } L \)
5. \( \text{if } x \text{ goto } L \ / \ \text{ifFalse } x \text{ goto } L \)
6. \( \text{if } x \text{ relop } y \text{ goto } L \)
7. function calls:
   - \( \text{param } x \)
   - \( \text{call } p, n \)
   - \( y = \text{call } p \)
   - \( \text{return } y \)
8. \( x = y[i] \) and \( x[i] = y \)
9. \( x = &y, x = *y, *x = y \)
"The description of three-address instructions specifies the components of each type of instruction, but it does not specify the representation of these instructions in a data structure."

[p. 366]
Quadruples

Instructions have four fields: 
\text{op}, \text{arg1}, \text{arg2}, \text{result}

Example: \( t_3 = a + t_2 \) is represented as

\[
\begin{array}{cccc}
\text{op} & \text{arg1} & \text{arg2} & \text{result} \\
+ & a & t_2 & t_3 \\
\end{array}
\]

Example: \( t_4 = -c \) is represented as

\[
\begin{array}{cccc}
\text{op} & \text{arg1} & \text{arg2} & \text{result} \\
\text{minus} & c & & t_4 \\
\end{array}
\]
Variables in representation

Identifiers would be pointers to symbol table entries. Compiler-introduced temporaries can be added to the symbol table.

<table>
<thead>
<tr>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
<th>result</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>→ entry for a</td>
<td>→ entry for t₂</td>
<td>→ entry for t₃</td>
</tr>
</tbody>
</table>
Instructions have three fields:

- $op$, $arg_1$, $arg_2$

Example:

$t_2 = ...$

$t_3 = a + t_2$

is represented as

<table>
<thead>
<tr>
<th>line</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>$+$</td>
<td>$a$</td>
<td>(5)</td>
</tr>
<tr>
<td>6</td>
<td>comp. of $t_2$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Indirect triples

Because order matters (due to embedded references instead of explicit variables) it is more challenging to rearrange instructions with triples than with quadruples.

Indirect triples allow for easier reordering (see page 369).
Indirect Triples

Instructions have three fields:

\[ \text{op, arg1, arg2} \]

Example:

\[ t_2 = \ldots \]
\[ t_3 = a + t_2 \]

is represented as

<table>
<thead>
<tr>
<th>index</th>
<th>instruction</th>
<th>line</th>
<th>op</th>
<th>arg1</th>
<th>arg2</th>
</tr>
</thead>
<tbody>
<tr>
<td>72</td>
<td>5</td>
<td>5</td>
<td>computation of ( t_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>73</td>
<td>6</td>
<td>6</td>
<td>+</td>
<td>( a )</td>
<td>((72)))</td>
</tr>
</tbody>
</table>

Rearranging instructions changes the instruction array contents, but the instructions themselves do not change.
Static Single Assignment (SSA) is an intermediate representation that facilitates certain code optimizations. 

1) Each variable is assigned to exactly once. Occurrences of the same variable are subscripted to make them unique.

\[
\begin{align*}
x &= r + 1 \\
y &= s \times 2 \\
x &= 2 \times x + y \\
y &= y + 1
\end{align*}
\]
Static Single Assignment (SSA)  
an additional constraint on the three address code

1) Each variable is assigned to exactly once.

2) Need $\phi$ function to merge split variables:

\[
\text{if (e) then } \{ x = a \} \text{ else } \{ x = b \} \\
y = x
\]

With SSA:
\[
\text{if (e) then } \{ x_1 = a \} \text{ else } \{ x_2 = b \} \\
y = \phi( x_1, x_2 )
\]
In $y = \phi(x_1, x_2)$ simply let $x_1$ and $x_2$ be bound to the same address.
Review

- Project 1: char stream $\rightarrow$ LEXER $\rightarrow$ token stream
- Project 2: PARSER builds symbol table, checks for undefined or multiply defined names from token stream.
- Project 3: PARSER will also perform type checking and generate intermediate code.
Our language
(use name equivalence)

- **pre-defined types:**
  - **primitive types:** integer, real, Boolean, character
  - **composite type:** string

- **user-defined types:**
  - **record types have names**
    - `type rec : [ real : x , y ]`
  - **array types have names**
    - `type arr : 2 -> string`
  - **function types have names**
    - `type fun : ( real : x ) -> rec`
Recursive records
Recursive functions

A record type must allow a component to be of the same type as the type itself:

type Node: [ integer datum:=0 ; Node rest:=null ]